POLITECNICO MILANO 1863

Model Checking of Battery-Powered Railway Lines

Formal Methods for Concurrent and Real-Time Systems

Andrea Carotti, Emilio Ingenito
MSc in Computer Science & Engineering

Prof. Pierluigi San Pietro
Dr. Livia Lestingi

A.Y. 21/22

Contents

1 Design

1.1 Line layout

In our model, we characterized the line as a list of stations and a list of trains. Due to the design choices we made, it is possible to implement **S** Stations and **T** Trains.

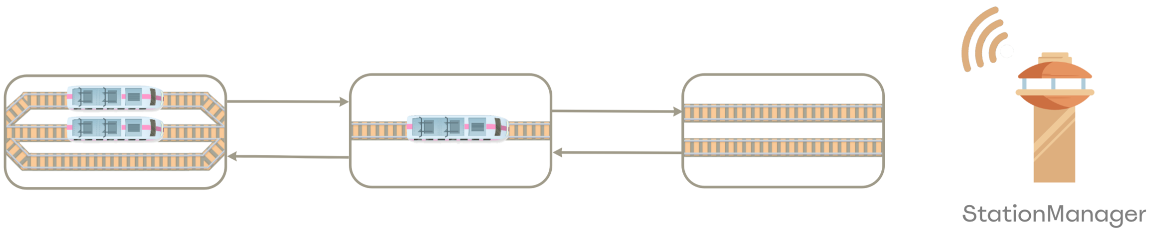

Figure 1: Layout of the System

1.2 Train logic

Together with the stations, trains are the core elements of the line. Trains travel between stations (loosing charge), and make requests to access them, or in the case they are not allowed, need to wait outside, loosing part of their charge. A train can be in one of the following main states:

- `traveling`: the train is traveling from one station to another, following the station's order as specified in the station vector;
- `waiting inside a station`:the train is waiting, recharging its batteries, inside its current station;
- `waiting outside a station`:the train, once arrived, is waiting for the station's approval, in order to get into it.

We considered different rate at which a train's charge drops: in detail, while traveling, the charge of the trains drops of a specific amount `CHARGE_DROP`; instead, while waiting outside a station, the the trains loose charge at a different, lower amount `CHARGE_DROP_WAIT`. On the other hand, while inside a station, the trains increase their charge by a given amount (`RECHARGE`).

1.3 Line logic

Every time a *train* reaches destination and it is waiting outside a *station*, it sends a request to enter to a specialized entity, the *StationManager*. Exploiting a set of matrices, it checks and keeps updated the status of the system (iterating over those matrices). Provided that certain policies are satisfied (e.g. the fact that there is enough space to host the waiting train), the StationManager lets the train enter the station: instead, if those conditions do not verify, it tells the train to stop and wait outside the station, where the train will loose charge and send back, each R time instants, the request to enter. When a train is ready to leave the station (e.g. it has enough time and charge to reach the next station and all the policies are satisfied), it waits for the StationManager's approval to exit: once got it, it starts its new travel. From a general perspective, most of the logic is concentrated in the Train and in the StationManager entities and, specifically, in their interaction (the access/exit to/from stations). The station, instead, will not cover a crucial role, and it's behaviour depends upon the directives provided by the previously described interaction between StationManager and Train.

1.4 Design assumptions

The following assumptions were made building the model:

- `unit of measure`: we assumed that lengths are expressed in *Units* and time is expressed in *Clocks*. This assumption allowed us to focus on the crucial aspects of the system, overshadowing real units of measure that can easily be taken into account at a later time;
- `train's speed`: train travel at constant speed, which is specified in a constant vector; (**trainVelocity**[**TrainNumber**])
- `maximum delay`: as suggested, we assumed that the maximum delay allowed between consecutive stations is the same in both directions, furthermore we took into account that the train's timer to reach the next station (initialized with the *maximum delay* value) starts as soon as the train enters inside a new station.

2 UPPAAL Model

2.1 Global Declarations

2.1.1 System Parameters

In order to handle the complexity of the system, we used both global vectors and matrices. Global vectors were useful to store general information that would be of interest of also other entities like the StationManager. Global matrices instead were used to map and bound each train with the station it is willing to communicate with.

Parameter	Description
`R`	Time between two consecutive requests to enter
`T`	Total number of trains
`S`	Total number of stations
`RECHARGE`	Amount of recharge while waiting inside a station
`CHARGE_DROP`	Amount of charge lost while travelling
`CHARGE_DROP_WAIT`	Amount of charge lost while waiting outside a station
`SAFE_TIME_MULT`	Custom Parameter for leaving the station policy

In the table above are shown the main parameters of the system, that can be set in order to obtain different configurations and, consequently, analysis results. By changing these parameters we can substantially modify our system's behaviour.

Data Structure	Description
`trainWantsToAccess[i][j]`	Matrix indicating whether in that time instant there is train j trying to enter in station i
`trainCanAccess[i][j]`	Boolean matrix indicating whether in that time instant the access to train j in station i was granted
`trainWantsToLeave[i][j]`	Boolean matrix indicating whether in that time instant there is train j trying to leave in station i
`trainCanLeave[i][j]`	Boolean matrix indicating whether in that time instant train j was approved to leave station i
`trainIsWaiting[i][j]`	Boolean matrix indicating the trains j that are waiting outside station i

2.1.2 Variables and channels

We model the interaction between trains and stations with the support of several data structures. Every variable starting with *train* is an array, the dimension depends on the total number of trains declared in the system. The reason of this choice is that every train can easily edit its status by checking on the array by using its unique ID, and at the same time, the StationManager can observe and edit the status of the trains and taking decision by iterating on that.
For Synchronization purposes we defined the following **channels**:

- `initialize:`Used to initialize the entire system
- `updateStatus:`Channel used to change the status of a station
- `go:` Used to synchronize all the actions while the line is working
- `waitingToEnter:`As soon as the train reaches the station and it is waiting outisde, it sync on this channel requesting to the StationManager, which checks if the train is allowed to enter or not.
- `waitingToLeave:`As soon as the train is able to leave the station checking all the proper parameters to do so, and it is waiting inside, it sync on this channel sending the request to the StationManager, which checks if the train is allowed to leave or not.
- `enter:`After checking whether the train could enter or not in the station, if the answer was positive the StationManager gives access to the train to enter (if many trains made the request only the train with the worst condition is allowed to enter).
- `leave:`After checking whether the train could leave or not the station, if the answer was positive the StationManager lets the train to leave (if many trains made the request it sends a random train among those satisfying the policies to leave).
- `wait:`Synchronizes two different situations: 1) If the train requested to enter in a station to the StationManager but was not allowed to do so. 2) If a train tried to leave a station but was not allowed to do so.

The synchronization in the system between many trains occurs through the go that works as a clock.

2.2 Templates

2.2.1 Initializer

This is a simple automaton which only has two transition, the first one executes immediately as the initial state is committed. The second one instead is used to synchronize the actions of the trains in the system. The initialize_all() function generates the starting configuration and initialize global variables. All the other automaton of the system are signaled to start through the initialization channel. In this way the whole system is set up at the same time.

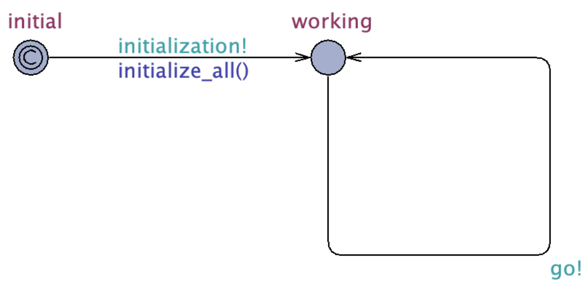

Figure 2: Initializer

2.2.2 Station

The Station behaviour is passive, since the only role in the system is to change state depending on the number of trains currently inside of it. It is a useful automaton for testing, to check the stations' status and the proper interaction with the StationManager.

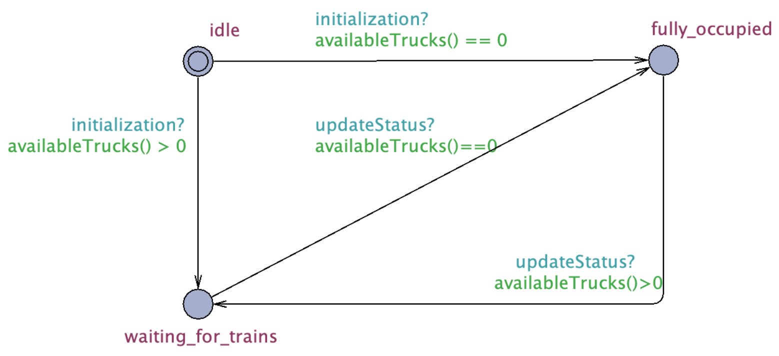

Figure 3: Station

2.2.3 StationManager

The StationManager has priority over the other Timed Automata. It handles the way a train enters and leaves the actual station and responds to the requests made by the trains. Hence, every time a train wants to enter into a station, it makes a request to the StationManager which works only in committed states and update the matrix that keeps track of which trains want to **enter/leave** which station. Each request of the trains is handled one at the time. If many trains want to enter, it chooses the train with the most critical situation (the one that is going to go out of time or charge before the others), allows it to enter and update the current number of available trucks in the station. If a train wants to leave a station, the request is handled similarly, updating the number of available trucks, and checking on the support matrix if the train has the ability to leave the station.

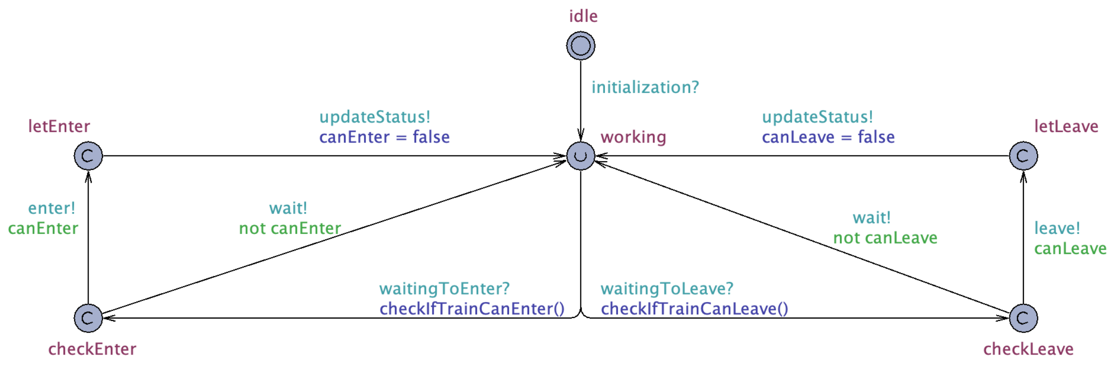

Figure 4: StationManager

2.2.4 Train

The train contains most of the logic in the system. If it's inside a station, every time instant it changes its charge, and reduces the time left to reach the next station, since, in our model the timer (delay from one station to one another) starts as soon as the train reaches a new station. It also checks whether it can leave the actual station: this is where the policies for leaving the station are applied. If the policies are met the train updates the status of the **matrix**, in order to communicate with the StationManager that it can leave the actual station: then, it waits for the StationManager to reply (on channel *wait*).

While traveling, the train updates its charge, its distance from the next station and the time left to reach the next station. At every instant it is checked whether its charge or its time left reached 0. As soon as the distance is 0, through channel *waitingToEnter* the Train sends the request to the StationManager, and as soon as the StationManager sync on that channel (it could be that was serving before another train), it goes on **ready_to_enter state**.

Depending on the answer of the StationManager the train can go to **waiting outside a station**, and decreases its charge until it can manage to make a new request after R time instances, or can go directly inside the station and update all the travel info.
The updated info are:

- `next station of the train`
- `actual station of the train`
- `train time left` (the new delay is triggered as soon as the train enters the station)
- `distance to next station`

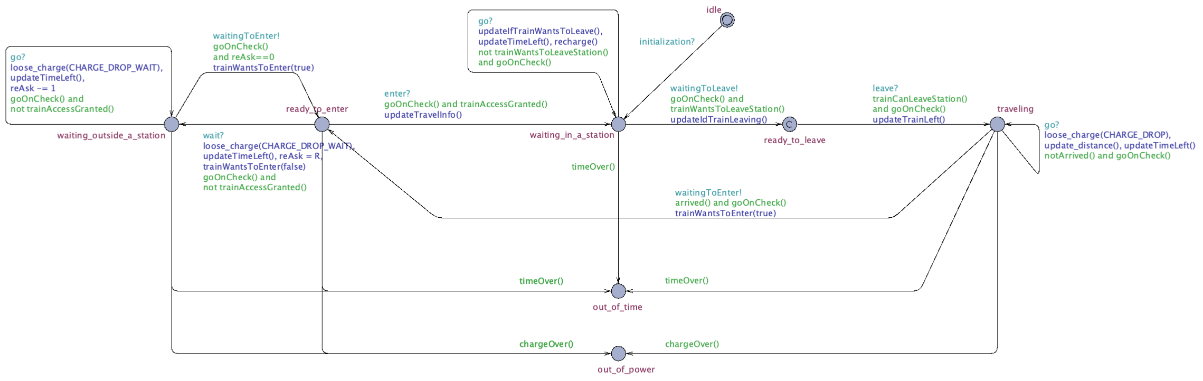

Figure 5: Train

3 Policies and queries

3.1 Policies

We designed two main policies to manage the time a train spends in a station: the crucial aspects, we focused on, were the charge and the time left, that should allow the train to, at least, travel to the next station. The first policy is called *Safe policy*, since it checks if the system is facing a critical situation, which is characterized by the following properties:

1. the train T is waiting inside the station S and is recharging its batteries
2. the station S has no more available trucks
3. at least one train is waiting outside of station S, willing to enter into it.

If this is the case, it is checked if there is at least one waiting train whose *conditions* (i.e. the current charge and time left) are worse than the current's ones. If so, the train T asks the StationManager to leave the station as soon as possible.

```
s = StationIndex;

bool checkSafePolicy(){
    bool waitingTrain;
    for (int i=0; i<T; i++){
        if(trainIsWaiting[s][i] == true)
            waitingTrain = true;
    }
    if(stationAvailableTracks[s] == 0 and waitingTrain) {
        if (trainWorseThanCurrent(s))
            return true;
    }
    return false;
}
```

Listing 1: Safe policy

The second policy (*Greedy policy*) focuses on the current train's status, and aims to maximize its charge. In order to succeed in this purpose, we have to constantly check the remaining time to reach the next station (which is a critical parameter). In this case, if the time left goes under a specific, let's say *safe*, amount of time, the train asks the StationManager to leave the current station.

```
t = TrainNumber;
expectedTime = trainDistanceToNextStation[t] / trainVelocity[t]

bool checkBestConditionsPolicy(){
    if ( trainTimeLeft[t] <= expectedTime * (SAFE_TIME_MULT/100))
        return true;
    else
        if (trainActualCharge[t] == trainMaxCharge[t])
            return true;
        else
            return false;
}
```

Listing 2: Greedy policy

Working together, those policies ensure that our system works properly, so each train reaches the next station in its *best conditions* and no train is left waiting outside a station if its conditions are critical.

3.2 Queries

We took into consideration some queries, with the purpose of verifying that our system would work properly even if stressed by the presence of bottlenecks. So, the first two (mandatory) queries verify that every train reaches the destination in time, and never goes out of power. The remaining queries try to check if there is at least one state in which the system works *under pressure*, i.e. a train goes under a certain amount of power or time left (to reach the next station).

There is no train whose Time Left to reach the next station drops to 0:

$$\forall \Box\, (\, \neg T0.out\_of\_time \;\wedge\; \neg T1.out\_of\_time \;\wedge\; \neg T2.out\_of\_time \;\wedge\; \neg T3.out\_of\_time)$$

There is no train whose charge drops to 0:

$$\forall \Box (\neg T0.out\_of\_charge \wedge \neg T1.out\_of\_charge \wedge \neg T2.out\_of\_charge \wedge \neg T3.out\_of\_charge)$$

There is at least one state where the train charge is under a certain value K:

$$\exists \Diamond\, (\, (T0.timeLeft < K) \;\vee\; (T1.timeLeft < K) \;\vee\; (T2.timeLeft < K) \;\vee\; (T3.timeLeft < K))$$

There is at least one state where where the charge of a train is under a certain value H:

$$\exists \Diamond\, (\, (T0.actualCharge < H) \;\vee\; (T1.actualCharge < H) \;\vee\; (T2.actualCharge < H) \;\vee\; (T3.actualCharge < H))$$

The number of available tracks is always less than the total number of tracks of the considered station

$$\forall \Box\, (\, (S0.availableTracks < S0.numberOfTracks) \;\wedge\; (S1.availableTracks < S1.numberOfTracks) \;\wedge\; (S2.availableTracks < S2.numberOfTracks) \;\wedge\; (S3.availableTracks < S3.numberOfTracks))$$

Workflow to develop the policies:

1. In the initial configuration we took into account (3 stations and 3 trains), in order to satisfy the mandatory queries it was sufficient to check whether the train had a charge over a certain percentage of its **maxCharge**, and whether it was able to travel for at least 1.5x the length of the distance to the next station. Making the system more complex, (i.e. adding trains and stations), we realized that such policies were too simple for the system to work properly.

2. So, we introduced that, if a train was in a station, and that station is a bottleneck (so currently no other trucks are available for incoming trains) and there's another train waiting outside the station, the current train should leave the station as soon as possible.

3. Finally we introduced a parameter `SAFE_TIME_MULT` based on the characteristics of the line (distances among stations, trains' velocities etc.). If the *trainTimeLeft* is less than this parameter multiplied by the time required to reach the next station, the train needs to leave; otherwise it is checked the charge, and if it is close to the **maxCharge**, then the train can leave. The reason for this choice is that while the charge always increases in stations by the same amount each time instant, the time delay depends from station to station so it needs a parameter to be modelled with.

4 Analysis and Results

In order to test and verify our system, we considered a specific configuration, which involved:

- `one line`
- `four trains`
- `five stations`

We set the distances between each pair of station, the maximum delay and the train velocities (these parameters are strongly correlated) to make our system look as realistic as possible. The following table summarizes the initial setting of the main variables.

TrainID	Actual S.	Next S.	Velocity	Charge	TimeLeft
`Train0`	`Station0`	`Station1`	`10`	`100`	`30`
`Train1`	`Station1`	`Station0`	`11`	`100`	`30`
`Train2`	`Station2`	`Station1`	`10`	`100`	`22`
`Train3`	`Station0`	`Station1`	`11`	`100`	`30`

4.1 First Scenario: Efficient Railway-Line

In order to choose the right parameters, we took into account that the system would be much more efficient if, considering similar speeds for trains, we would increase distances and lower the allowed delay between stations. The figure below shows the distances between each pair of stations and the corresponding maximum delay. Slightly changing those parameters will not compromise the proper behaviour of the system, highlighting that this configuration is not pushing the system to its limits.

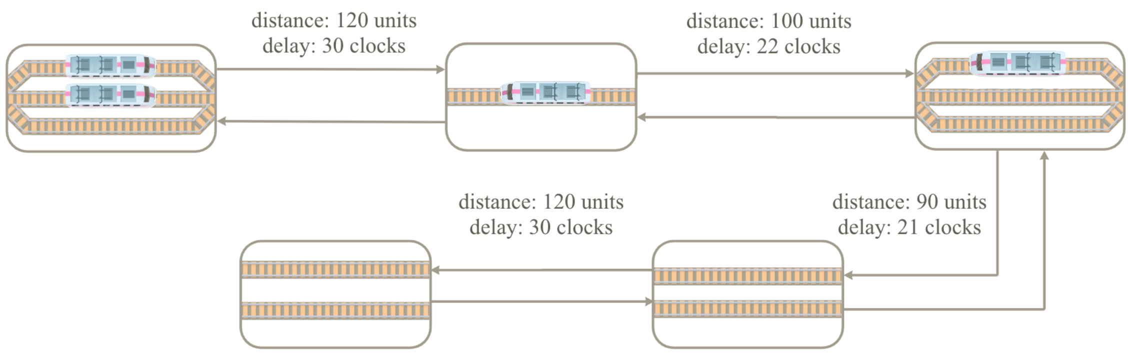

Figure 6: Layout of the Efficient configuration

4.2 Second Scenario: Inefficient Railway-Line

With the idea of analyzing a possible inefficient scenario, we reconfigured the system, changing most of the cruial parameters, such as:

- `recharge amount`: by dropping it, we could effectively ascertain that the charge of train's battery, gained during the time spent in the stations, was not enough;
- `charge drop amount`: by raising this parameter, trains will loose more charge while traveling;
- `safe time multiplier`: by dropping this value, the trains will have less margin to reach the following station;
- `distances between stations` : by increasing them, without increasing the maximum delay, we stressed even more the system.

The result was a system in which the trains were not able to reach the following station in time, and their charge could go below 0%.

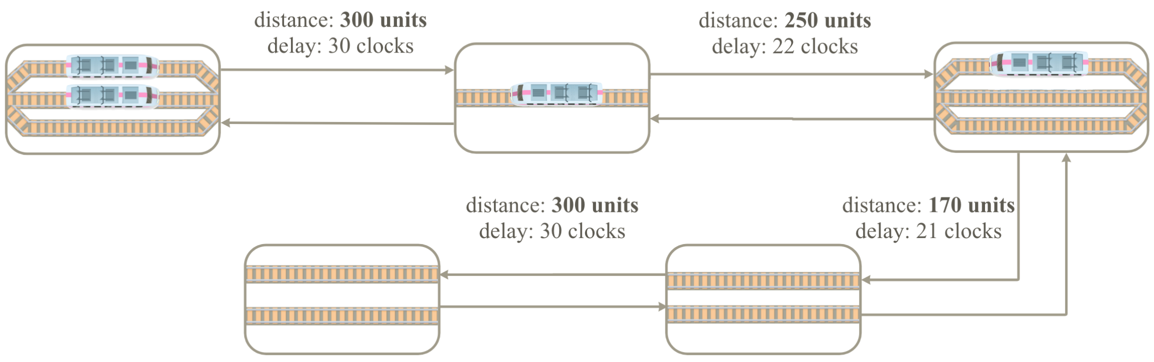

Figure 7: Layout of the Inefficient Configuration

5 Conclusions

To summarize the work done, the model built was made having in mind to extend the possible number of stations and the total number of trains. A great part of the effort was spent to understand how to manage multiple trains requesting access to multiple stations, and to build efficient policies due to the vast number of parameters that can vary in the system (speed of trains, distance between stations, total charge of trains, delay between the stations). In order to reach a good efficiency for the model we used a parameter chosen taking into account the dependencies between velocity, distance and delay.